Overview of the approach used

The approach used: Event-Driven Approach

Our main approach for the project is to create our pipeline which is where we send data from our source to a target as data happens in real-time as streams of events(data). This approach of creating and using a pipeline will help us get more accurate and more up to data in the target system. Also, we only have to wait until data is propagated through the pipeline and get to react to data as it changes. Additionally, this enables you to get more economical use of computing and storage resources.

In regards to Kafka streaming data, the pipeline is means of ingesting the data from some source system into Kafka as the data is created. And then streaming that data from Kafka to one or more target systems. Because Kafka is a distributed system it is quite scalable and resilient and serves as a distributed log(data storage system) here. Some of the benefits of decoupling the source system from the target system using Kafka in our pipeline are:

- Kafka can store our data as long as we want: so if the target system goes down there will not be any long-term permanent impact on our pipeline. Also if for any reason the source system goes offline there won't be any issues to our pipeline and the target system may not even know it's actually offline, there just might be less traffic or no traffic due to the source system being offline. It will resumes from the point it went down without any problem when it's back up.
- Pipelines created with Kafka can evolve smoothly a piece at a time: since the data stored by Kafka can be sent to multiple targets independently.

In our pipeline, we will have a:

- Text selector(producer) that will input text corpus from S3 and output a single selected text to Kafka topic1.
- the Kafka topic1 has single selected text input and outputs the text upon request
- On the Web application, the consumer will view the selected text
- On the Web application, the producer will create an audio file from the user and that goes to a Kafka topic2
- Kafka topic2 will have an audio file input and output it upon a request

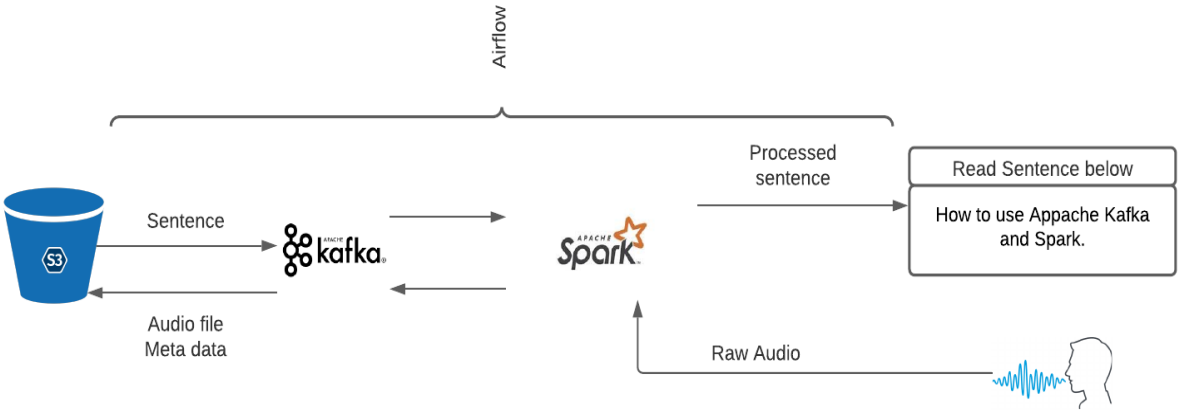
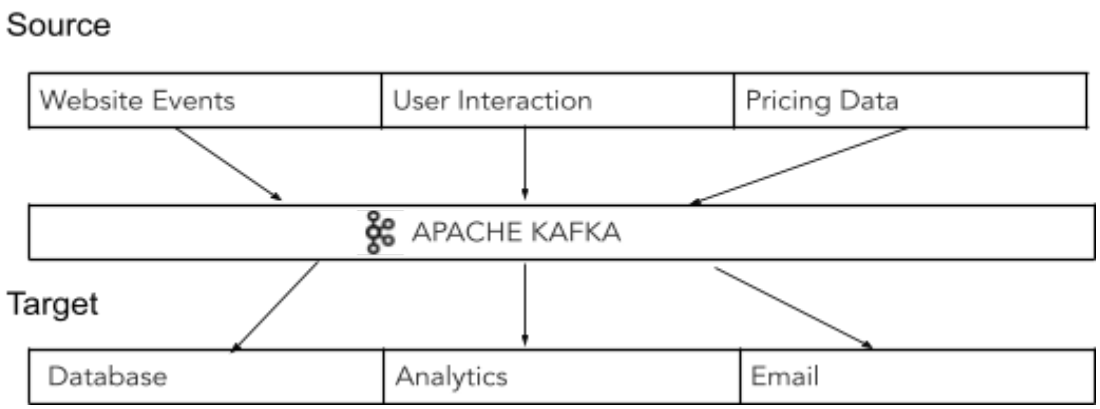


diagram of our pipeline overview

Why use Apache Kafka?

Kafka is a distributed streaming(continuous flow of data or instructions) platform. It's used as a transportation mechanism to move data real fast in scale. It addresses the complexity issue and increased load when having many source systems and target systems exchanging data. Kafka has high-performance meaning latency of less than 10 ms and high throughput as well.

1. It provides abstractions to deal with streams of events in a reliable, fault-tolerant manner.
2. Kafka gives real-time insights to users
3. Storage capability.
4. Uses log data structure architecture(appends - not deleted or updated)
5. Apache Kafka allows you to decouple your data stream & systems
    a. Source system- contains data in Kafka
    b. Target system - sources data from Kafka



6. Kafka provides two APIs
    1. Producer API- used to publish events to topics
    2. Consumer API - used to subscribe to topics and process their messages on the fly.

Generic term	Kafka equivalent
Event	Message
Stream	Topic
Publisher	Producer
Subscriber	Consumer

In Kafka, we have topics that are a particular stream of data similar to a table in the database and can have many of them. These topics are split into partitions which are part of topics and get incremental id called offsets. Once data is written its immutable and data is assigned randomly to a partition unless a key is provided. Kafka cluster is composed of many brokers(servers) each having a unique id. Once connected to one broker you can have access to other brokers in the cluster. Topics should have a replication factor of at least 2 so that if one server(broker) is down the others can serve the whole data. Additionally, at any time only one leader for a given partition which means only that leader usually indicated by a start can receive and serve data. The rest of the brokers are in-sync replicas (ISR). The brokers will be managed by Zookeepers. When it comes to producers they are to write data on topics. Producers can choose acknowledgments:

Ack = 0: producers won't wait for any acknowledgment (possibility of data loss)

Ack =1: producers will wait for leader acknowledgment(limited data loss)

Act = All: producers will wait for both leader and replicas acknowledgment

Tools to be used:

1. Kafka streams
2. KSQL by confluent

Why use Airflow?

Apache Airflow is an open-source scheduler to manage your regular jobs. It is an excellent tool to organize, execute, and monitor your workflows so that they work seamlessly.

DAGs (Directed Acyclic Graphs) represent a workflow in Airflow. Each node in a DAG represents a task that needs to be run. The user mentions the frequency at which a particular DAG needs to be run. The user can also specify the trigger rule for each task in a DAG. e.g., You may want to trigger an alert task right after one of the previous tasks fails.

Components of Airflow

1. Scheduler- It is responsible for scheduling your tasks according to the frequency mentioned. It looks for all the eligible DAGs and then puts them in the queue.
2. Web Server - The web server is the frontend for Airflow. Users can enable/disable, retry, and see logs for a DAG all from the UI. Users can also drill down in a DAG to see which tasks have failed, what caused the failure, how long did the task run for, and when was the task last retried.
3. Executor - It is responsible for actually running a task. The executor controls on which worker to run a task, how many tasks to run in parallel and updates the status of the task as it progresses.
4. Backend - Airflow uses MySQL or PostgreSQL to store the configuration as well as the state of all the DAG and task runs. By default, Airflow uses SQLite as a backend by default, so no external setup is required. The SQLite backend is not recommended for production since data loss is probable.

So, why use Airflow?

The scheduler of the data pipeline.

- For monitoring
    - In case we want to see the status of a task from UI.
    - It sends an email in case a DAG fails.
    - We can also view the logs for a task from Airflow UI
- Lineage
    - allows us to track the origins of data, what happens to it, and where it moves over time, such as Hive table or S3/HDFS partition.
    - To define the input and output data sources for each task. A graph is created in Apache Atlas, depicting the relationship between various data sources.

Why use Spark?

Spark is a general-purpose distributed data processing engine. Apache Spark is a lightning-fast cluster computing designed for fast computation. We will incorporate Spark into our application to rapidly query, analyze, and transform data at scale. Spark Structured Streaming makes it easy to build streaming applications and pipelines with the same and familiar Spark APIs.

The main feature of Spark

In-memory cluster computing increases the processing speed of an application.

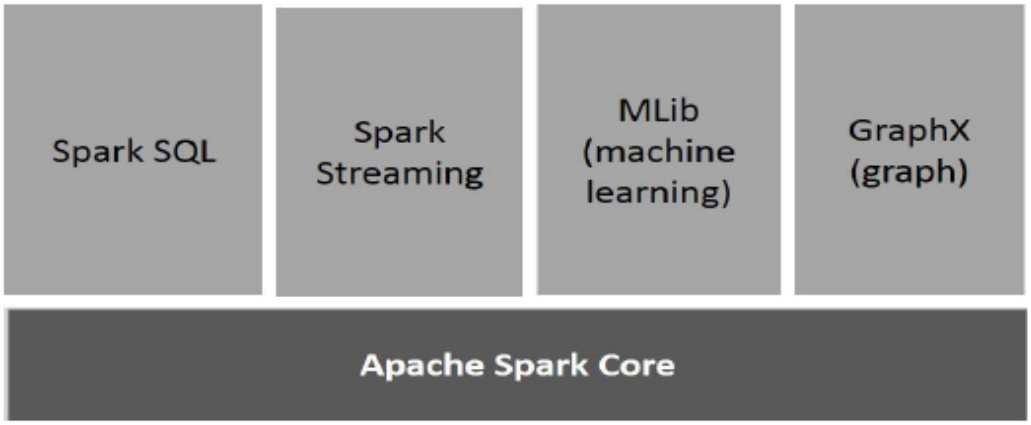
Spark use case:

1. Stream processing- The streams of data can be processed in real-time
2. Interactive query process that is speed efficient
3. Data Integration- used to reduce the cost and time required for this ETL process

Spark also provides a Python API. To run Spark interactively in a Python interpreter, use bin/pyspark:

```
./bin/pyspark --master local[2]
```

Components of Spark



How the approach would provide a good data source to the clients

Kafka clusters contain multiple brokers. The partitions of a particular topic are distributed among several brokers to aid in horizontal scaling. From the user's (The individual who records audio) point of view, different consumers can access different instances of topic partition. They can therefore upload files in parallel without lagging the system. This would therefore improve the user experience while interacting with the system

From the receiver of the audio point of view, data ETL would be real-time due to the capability built on Kafka. Once a user uploads an audio file equivalent to the text, it is immediately sent to the Kafka cluster, then gets transformed by Spark before being loaded into the data lake. This approach ensures timely access to important data and would provide an efficient platform for Continuous Integration and Continuous Development. The CI/CD would be enabled In such a way that the clients can start developing products with the data available and incrementally fine-tune the system as more data gets continuously loaded into the system.